http://www.microsoft.com/spanish/msdn/articulos/archivo/260402/voices/errhandlvbnet.asp

Control de errores en Visual Basic .NET

Actualización a Microsoft .NET

Ken Getz MCW Technologies

Febrero de 2002

Resumen: en este artículo se describen las diferencias entre el sistema de control de errores de Visual Basic .NET y el utilizado en Visual Basic 6.0. Entre los temas principales que se tratan en este documento se incluyen los bloques Try/Catch, los objetos Exception, los llamadores de procedimientos y la creación de clases de excepciones propias. (17 páginas impresas.)

Objetivos

- Comparar el sistema de control de errores de Microsoft® Visual Basic® .NET con el de Visual Basic 6.0
- Aprender a utilizar los bloques Try/Catch para controlar los errores en tiempo de ejecución
- Utilizar los objetos Exception para determinar el tipo de error producido
- Devolver excepciones a los llamadores de procedimiento
- Crear clases Exception propias

Contenido

Introducción al control de excepciones estructurado

.NET Framework proporciona en Visual Basic .NET un sistema de control de excepciones estructurado basado en el uso de las palabras clave **Try**, **Catch**, **Finally** y **Throw**. Este tipo de control de errores ya está disponible en C++ desde hace varios años. No obstante, desde la aparición de .NET Common Language Runtime, este sistema se puede encontrar en todos los lenguajes .NET, incluido Visual Basic .NET.

Antes del control de excepciones estructurado

Aunque Visual Basic dispone de su propio mecanismo de control de errores desde que el término "Visual" se agregó al nombre del producto, las técnicas ofrecidas a los programadores de este lenguaje presentaban demasiadas lagunas. Existen varios aspectos en torno al sistema de control de errores de Visual Basic (consulte la lista 1) que han sido motivo continuo de queja entre los programadores, tanto con experiencia como sin ella:

- En Visual Basic 6.0 es necesario realizar saltos dentro de los procedimientos para llevar a cabo el control de errores. Las instrucciones **On Error Goto**, **Resume** y **Resume Next** implican saltos en el código, tanto hacia adelante como hacia atrás. Las técnicas estándar de control de errores de Visual Basic 6.0 implican, al menos, un salto dentro del procedimiento y, a menudo, varios (uno hacia adelante, hacia el bloque de control de errores, y otro hacia atrás, hacia el punto de salida de un procedimiento común).
- De acuerdo con las instrucciones de programación de Visual Basic 6.0, en las que se recomienda, por ejemplo, que se garantice que los procedimientos dispongan de un único punto de salida, el lugar más adecuado para dicho punto es en mitad del procedimiento (antes del bloque de control de errores). Esto conlleva que, a menudo, se olviden las importantes instrucciones **Exit Sub** o **Exit Function**.
- En Visual Basic 6.0 no hay forma de *insertar* y *extraer* controladores de errores. Para conservar la interceptación del error actual, debe configurar un controlador diferente y, a continuación, volver al primero. No olvide incluir la instrucción **On Error Goto…** adecuada cada vez que cambie de controlador.
- Visual Basic 6.0 incluye un único objeto **Err**, por lo que, si se produce un error y no lo controla inmediatamente, puede perder permanentemente la información relativa al mismo.
- En la documentación de Visual Basic 6.0 apenas se incluye información sobre los tipos de errores (es decir, el número de errores) que se pueden generar como consecuencia de la realización de una acción determinada en el código. La única alternativa es intentar generar la mayor cantidad de números de error desencadenando dichos errores e interceptar aquéllos que sean específicos de su código.

Lista 1. El control de errores de Visual Basic 6.0 requiere, al menos, un salto y, a menudo, varios.

```
Sub TestVB6()
    On Error GoTo HandleErrors

    ' Realizar una acción aquí que
    ' pueda generar un error.

ExitHere:
    ' Ejecutar el código de limpieza aquí.
    ' Omitir los errores en este
    ' código de limpieza.
    On Error Resume Next
    ' Ejecutar el código de limpieza.
    Exit Sub

HandleErrors:
    Select Case Err.Number
        ' Agregar casos para cada
        ' número de error que se desea interceptar.
        Case Else
            ' Agregar el controlador de errores final.
            MsgBox "Error: " & Err.Description
    End Select
    Resume ExitHere
End Sub
```

Asimismo, aunque los programadores de Visual Basic disponían del método **Err.Raise** para devolver los errores a los procedimientos que realizan la llamada, esta técnica nunca llegó a ser estándar. Gran

parte de los programadores de código llamado por terceros, en lugar de emitir un mensaje sólo en caso de error, devuelven un valor para indicar que la operación se realizó o no correctamente. Debido a que se pueden omitir fácilmente los valores de error devueltos de los procedimientos a los que se llama, en demasiadas ocasiones el código no devuelve el error en tiempo de ejecución adecuado a los llamadores correspondientes.

Después del control de excepciones estructurado

La incorporación del control de excepciones estructurado facilita a los programadores la administración de la notificación y generación de errores, así como la determinación de las causas de los mismos en tiempo de ejecución. Este sistema incluye una serie de características que lo convierten en un mecanismo de control más flexible que el utilizado en las versiones anteriores de Visual Basic:

- El sistema de control de errores de .NET se basa en la clase Exception, que contiene información sobre el error actual, así como la lista de errores que lo puede haber desencadenado.
- Puede heredar de la clase Exception, creando sus propias excepciones con la misma funcionalidad que la clase base. Asimismo, si es necesario, puede crear funcionalidad extendida. La creación de su propia clase Exception permite a su código interceptar excepciones específicas, lo que aporta un alto nivel de flexibilidad.
- Debido a que las clases de .NET Framework inician excepciones al encontrarse con errores en tiempo de ejecución, los programadores adquieren el hábito de interceptar y controlar las excepciones. De este modo, aumentan las posibilidades de que se realice con éxito el control de las excepciones iniciadas desde los componentes.
- Puede anidar los bloques Try/Catch con los bloques **Try**, **Catch** o **Finally**, lo que permite a los programadores administrar el control de errores en el nivel de granularidad requerido.

En la lista 2 se muestra el diseño de un controlador de excepciones simple en Visual Basic .NET. En las siguientes secciones se describe en detalle el uso de cada una de las palabras clave incluidas en la lista 2, así como la utilización de la clase Exception en el seguimiento y generación de errores.

Lista 2. El control de errores en Visual Basic .NET no requiere saltos.

```
Sub TestVBNET()
  Try
    ' Realizar una acción aquí que
    ' pueda generar un error.
  Catch
    ' Controlar excepciones que ocurren en
    ' el bloque Try aquí.
  Finally
    ' Ejecutar el código de limpieza aquí.
  End Try
End Sub
```

> **Sugerencia** Puede utilizar el antiguo mecanismo de control de errores de Visual Basic 6.0 junto con el control de excepciones estructurado de .NET en el mismo proyecto pero no dentro del mismo procedimiento. **On Error** y **Try** no se pueden incluir en el mismo

procedimiento.

Generación de errores

Los siguientes ejemplos comparten la misma premisa: el objetivo es abrir un archivo, recuperar su longitud y, a continuación, cerrarlo. Todos los ejemplos utilizan el código *txtFileName* para recuperar el nombre del archivo de un cuadro de texto del formulario de ejemplo:

```
Dim lngSize As Long    ' La longitud es un número de 64 bits.
Dim s As FileStream

s = File.Open(txtFileName.Text, FileMode.Open)
lngSize = s.Length
s.Close()
```

Por supuesto, el código puede generar (por varias razones) un error. Por ejemplo, se puede generar una excepción si:

- El archivo no se encuentra.
- La ruta no existe.
- La unidad en la que se ubica el archivo no está lista (tal vez ha solicitado el tamaño de un archivo en una unidad de disquete sin medio).
- No dispone de permiso para tener acceso al archivo o a la carpeta solicitados.
- Ha especificado un nombre de archivo no válido.

Esta lista podría seguir de forma indefinida. En los siguientes ejemplos se incluye una serie de variaciones en el código a fin de mostrar algunas de las características del control de excepciones estructurado.

Adición de características de control de errores

Los ejemplos que se ofrecen en las siguientes secciones incluyen características de control de errores cada vez más complejas en el código de error anterior e introducen los conceptos de interceptación e identificación de excepciones en Visual Basic .NET, partiendo del caso en el que no se ha incluido ningún código de control de errores.

La aplicación de ejemplo *ErrorHandling.sln*, incluye un formulario, *frmErrors*, que permite hacer uso de todas las técnicas descritas en este artículo (consulte la figura 1). En cada caso, escriba la ruta a un archivo o unidad que no exista, a una unidad que no contenga ningún medio o a cualquier otra ruta que pueda desencadenar un error en el sistema de archivos.

Figura 1. Utilice este formulario de ejemplo para probar todas las características aquí descritas.

Caso base: No hay código de control de errores

¿Qué ocurre si no se incluye código de control de errores? En ese caso, los errores que se producen en tiempo de ejecución vuelven al tiempo de ejecución de .NET. A continuación, aparece el cuadro de diálogo que se muestra en la figura 2, un cuadro confuso y que puede resultar peligroso. Para que no aparezca este cuadro de diálogo cuando se produzca un error en tiempo de ejecución, debe agregar código de control de errores, al menos a los procedimientos de nivel superior, así como a los de nivel inferior según sea necesario.

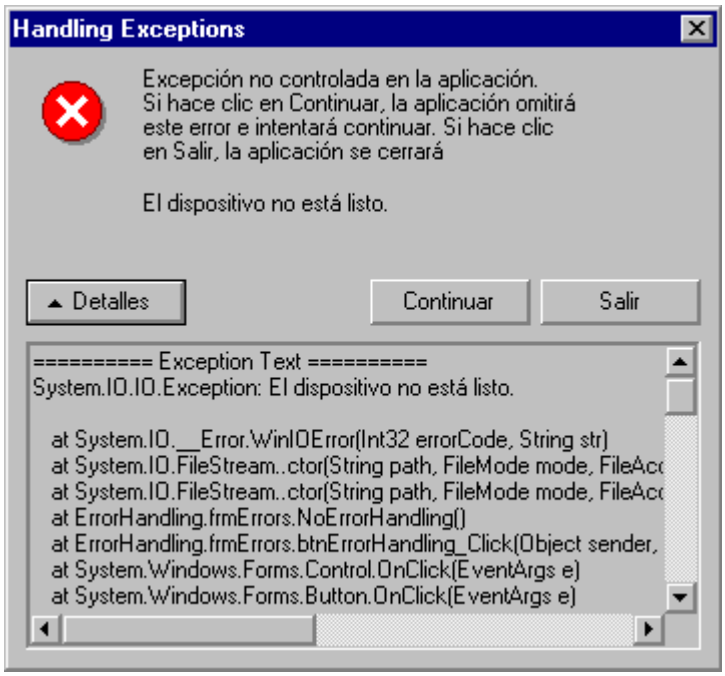

Figura 2. La adición de un botón Continuar hace que el controlador de errores predeterminado de .NET resulte un tanto peligroso. Asimismo, el usuario no tiene por qué ver los detalles del error.

> **Sugerencia** Al igual que en Visual Basic 6.0, si no agrega código de control de excepciones a un procedimiento determinado, y se produce un error en dicho procedimiento, el tiempo de ejecución de .NET extraerá el procedimiento actual de la pila de llamadas y volverá al procedimiento anterior. Si el procedimiento contiene código de control de errores, el tiempo de ejecución utilizará dicho código. Si no es así, el tiempo de ejecución continuará extrayendo procedimientos de la pila hasta que encuentre uno que contenga código de control de errores. Si ninguno de los procedimientos contiene código de control de errores, será el propio tiempo de ejecución de .NET el que se encargue de controlar el error, como se muestra en la figura 2.

Adición de un bloque Try/Catch/End simple

Para controlar correctamente los errores en tiempo de ejecución, debe agregar un bloque **Try**/**Catch**/**End Try** al código que desea proteger. Si se produce un error en tiempo de ejecución en el código del bloque **Try**, la ejecución continuará inmediatamente con el código del bloque **Catch**:

```
Try
  s = File.Open(txtFileName.Text, FileMode.Open)
  lngSize = s.Length
  s.Close()
Catch
  MessageBox.Show("Error")
End Try
```

Cuando se ejecuta este código, en lugar de que la aplicación muestre un mensaje de alerta y se detenga, el sistema emite un mensaje de alerta simple, "Error", y la aplicación continúa. Para probar esto, seleccione la opción **Simple Catch** en el cuadro combinado de control de errores del formulario de ejemplo.

> **Sugerencia** Si agrega un bloque **Try**/**Catch**/**End Try** al procedimiento, deberá incluir, al menos, un bloque **Catch** (más adelante encontrará información sobre la forma de incluir varios bloques **Catch**). Si desea omitir los errores, basta con no incluir nada en el bloque **Catch**. Aunque no es una idea demasiado acertada, le permitirá omitir los errores que puedan ocurrir.

Determinación de la causa del error

¿Cómo determinar la causa y el modo de controlar un error en tiempo de ejecución? Tiene varias opciones. Puede crear una variable, declarada utilizando **As Exception**, para recuperar la información del error. La clase Exception proporciona información sobre el error en tiempo de ejecución, como se muestra en la tabla 1.

Miembro	**Descripción**
HelpLink	Vínculo al archivo de ayuda asociado a esta excepción.
InnerException	Referencia a la excepción interna: la excepción que ocurrió originariamente, si esta excepción está basada en una anterior. Las excepciones se pueden anidar. Es decir, cuando un procedimiento inicia una excepción, dicho procedimiento puede anidar otra excepción dentro de la que está generando y pasar ambas al llamador. La propiedad **InnerException** proporciona acceso a la excepción interna.
Message	Texto del mensaje de error.
StackTrace	Seguimiento de la pila, en forma de cadena única, en el punto en el que se produjo el error.
TargetSite	Nombre del método que generó la excepción.
ToString	Convierte el nombre y la descripción de la excepción, así como la descarga actual de la pila, en una única cadena.

Message	Devuelve una descripción del error.

Tabla 1. Miembros útiles de la clase Exception

El bloque **Catch** incluye la referencia a la variable del siguiente modo:

```
Try
    ' Código que puede desencadenar una excepción.
Catch e As Exception
    ' Controlar la excepción, utilizando "e", aquí.
End Try
```

Asimismo, puede declarar la variable **Exception** fuera del bloque **Catch**:

```
Dim e As Exception
Try
    ' Código que puede desencadenar una excepción.
Catch e
    ' Controlar la excepción, utilizando "e", aquí.
End Try
```

Excepciones

Tipo de excepción	Condición
Security Exception	El llamador no tiene los permisos necesarios.
ArguementException	La *ruta* está vacía, contiene sólo espacios en blanco o incluye caracteres no válidos.
FileNotFoundException	No se encuentra el archivo.
ArguementNullException	La *ruta* o *modo* es una referencia nula (Nothing en Visual Basic).
UnauthorizedAccessExpection	La *ruta* es de sólo lectura o un directorio.
DirectoryNotFoundException	No se encuentra el directorio.

Puede utilizar el siguiente código para interceptar una excepción y mostrar texto en el que se indique el problema:

```
' Opción Simple Exception del formulario de ejemplo.
Private Sub SimpleException()
  Dim lngSize As Long
  Dim s As FileStream

  ' Mostrar todo el contenido del objeto Exception.
  Try
    s = File.Open(txtFileName.Text, FileMode.Open)
    lngSize = s.Length
    s.Close()
  Catch e As Exception
    MessageBox.Show(e.ToString)
  End Try
End Sub
```

Sugerencia El nombre del objeto **Exception** no es importante. El código de ejemplo utiliza *e* como nombre de la variable. Se trata simplemente de una decisión arbitraria. Si

```
    lngSize = s.Length
    s.Close()
  Catch e As Exception
    MessageBox.Show("Error ocurred: " & e.Message)
  End Try
End Sub
```

Hasta ahora hemos visto cómo interceptar una excepción en el momento en el que se genera e indicar al usuario la causa que la originó. No obstante, en la mayoría de las ocasiones también deberá determinar la acción que se debe tomar en función del error específico. En Visual Basic 6.0, esto conllevaba la adición de un bloque **Select Case** basado en el número de error activo. En Visual Basic .NET, por el contrario, es necesario agregar bloques **Catch** adicionales para cada uno de los errores que se deseen interceptar. En la siguiente sección se describe el modo de agregar esta funcionalidad a los procedimientos.

Uso de excepciones específicas

.NET Framework ofrece un gran número de clases de excepciones específicas. Todas estas clases heredan de la clase base Exception. En la documentación de .NET Framework se incluye una serie de tablas en las que encontrará todas las excepciones que se pueden generar al llamar a un método determinado. Por ejemplo, la figura 3, procedente de dicha documentación, le ayudará a determinar la causa del error durante la llamada al método **File.Open**.

Excepciones

Tipo de excepción	Condición
Security Exception	El llamador no tiene los permisos necesarios.
ArguementException	La *ruta* está vacía, contiene sólo espacios en blanco o incluye caracteres no válidos.
FileNotFoundException	No se encuentra el archivo.
ArguementNullException	La *ruta* o *modo* es una referencia nula (Nothing en Visual Basic).
UnauthorizedAccessExpection	La *ruta* es de sólo lectura o un directorio.
DirectoryNotFoundException	No se encuentra el directorio.

Figura 3. La documentación de .NET incluye todas las excepciones que se pueden generar al llamar al método File.Open

Los procedimientos pueden incluir tantos bloques **Catch** como sean necesarios para controlar las excepciones de forma diferente e individual. El siguiente procedimiento del proyecto de ejemplo comprueba varias excepciones diferentes y las controla de forma individual. Pruebe este procedimiento utilizando varias excepciones específicas. Por ejemplo, cambie el nombre del archivo de modo que se encuentre:

- En una ruta válida pero seleccione un archivo que no existe.
- En una unidad que no existe.
- En una ruta que no existe.

- En una unidad que no está lista.

```
' Opción Multiple Exceptions del formulario de ejemplo.
Private Sub MultipleExceptions()
  Dim lngSize As Long
  Dim s As FileStream

  Try
    s = File.Open(txtFileName.Text, FileMode.Open)
    lngSize = s.Length
    s.Close()
  Catch e As ArgumentException
    MessageBox.Show( _
     "El nombre de archivo especificado no es válido. " & _
     "Compruebe que no ha incluido sólo espacios.")
  Catch e As FileNotFoundException
    MessageBox.Show( _
     "No se puede encontrar el archivo especificado. " & _
     "Vuelva a intentarlo.")
  Catch e As ArgumentNullException
    MessageBox.Show("Se ha pasado un argumento Null.")
  Catch e As UnauthorizedAccessException
    MessageBox.Show( _
     "Se ha especificado un nombre de carpeta, no de archivo.")
  Catch e As DirectoryNotFoundException
    MessageBox.Show( _
     "La carpeta especificada no existe " & _
     "o no se puede encontrar.")
  Catch e As SecurityException
    MessageBox.Show( _
     "No tiene derechos suficientes " & _
     "para abrir el archivo seleccionado.")
  Catch e As IOException
    ' Controlador de excepciones genérico para errores de E/S
    ' todavía no detectado. En este caso, la causa debe de
    ' ser que la unidad no está lista.
    MessageBox.Show( _
     "La unidad seleccionada no está lista. " & _
     "Compruebe que la unidad contiene un medio válido.")
  Catch e As Exception
    MessageBox.Show("Error desconocido.")
  End Try
End Sub
```

Determinación de la jerarquía de excepciones

Los vínculos que se incluyen en la tabla de excepciones de la figura 3 le muestran documentación sobre el objeto Exception. En ella se incluye una jerarquía de herencia, como muestra la figura 4. Debe tener presente esta jerarquía a la hora de agregar varios bloques **Catch**.

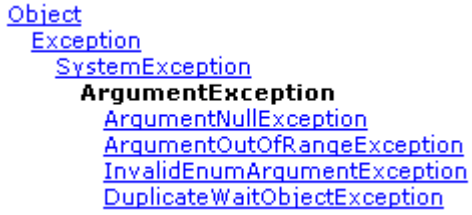

Figura 4. La jerarquía de herencia permite determinar la relación "es una" de los objetos.

Uso de la jerarquía de herencia de excepciones

En la jerarquía de excepciones que se muestra en la figura 4, se puede observar que ArgumentNullException hereda de ArgumentException que, a su vez, hereda de SystemException que, a su vez, lo hace de Exception. Cada nivel de la jerarquía indica un mayor nivel de especificación. Es decir, cuanto más inferior es el nivel de jerarquía, más específica será la excepción.

Debido a que cada nivel hereda de la clase definida sobre él, cada uno de los niveles inferiores constituye un ejemplo del tipo especificado. De este modo, la excepción ArgumentNullException "es una" excepción del tipo ArgumentException que, a su vez, "es una" excepción del tipo SystemException que, a su vez, "es una" excepción del tipo Exception. "es una" aparece entre comillas porque se trata de un operador significativo: cuando se dispone de varios bloques **Catch**, dichos bloques coinciden con la excepción actual utilizando la regla "es una". Es decir, al procesar varios bloques **Catch**, cuando el tiempo de ejecución encuentra la primera coincidencia de la excepción actual con la regla "es una" de la excepción interceptada por el bloque **Catch**, dicho tiempo de ejecución utiliza el bloque **Catch** para procesar la excepción y deja de buscar. En otras palabras, el orden de los bloques **Catch** es importante y se basa en la relación "es una". Todas las excepciones heredan de la clase base Exception, de modo que siempre se debe incluir un bloque **Catch** que controle dicha clase al final, si es que desea incluirla.

Generación de errores

Tal vez desee extraer los errores de los procedimientos para indicar a los llamadores que se ha producido una excepción. Es probable que sólo desee pasar una excepción en tiempo de ejecución estándar proporcionada por .NET Framework, o bien, crear su propia condición de excepciones. En cualquier caso, debe utilizar la palabra clave **Throw** para extraer la excepción del bloque actual.

> **Nota** El funcionamiento de la palabra clave **Throw** es muy similar al del método **Err.Raise** de Visual Basic 6.0.

Uso de la palabra clave Throw

Puede utilizar la palabra clave **Throw** de dos modos.

1. Devolver el error que acaba de ocurrir en el llamador desde un bloque **Catch**:

```
Catch e As Exception
  Throw
```

2. Generar un error desde cualquier código, incluido un bloque **Try**:

```
Throw New FileNotFoundException()
```

> **Nota** La primera técnica, iniciar la excepción que acaba de ocurrir, sólo funciona en los bloques **Catch**. La segunda de ellas, generar un nuevo error, funciona en cualquier bloque.

Búsqueda de controladores

Al iniciar una excepción, el tiempo de ejecución de .NET recorre la pila de llamadas a procedimientos en busca de un controlador de excepciones adecuado. (Si, cuando inicia la excepción, se encuentra en un bloque **Try**, el tiempo de ejecución utilizará los bloques **Catch** locales, si existe alguno, para controlar la excepción en primer lugar.) Cuando el tiempo de ejecución encuentra un bloque **Catch** de la excepción iniciada, ejecuta inmediatamente el código encontrado en dicho bloque. Si no encuentra ningún bloque **Catch** adecuado en la pila de llamadas, será el propio tiempo de ejecución el que se encargue de controlar la excepción (como se muestra en la figura 2).

Opciones del control de errores

Puede determinar las excepciones que desea controlar, así como las que quiere devolver a sus correspondientes llamadores. Cuando se produce una excepción puede:

- **No hacer nada.** En este caso, el tiempo de ejecución de .NET devolverá automáticamente la excepción al procedimiento que llamó al código.
- **Detectar errores específicos.** En este caso, no se devolverán las excepciones controladas. Sin embargo, las que no controle se devolverán al procedimiento de llamada.
- **Controlar todos los errores.** Agregue un bloque "Catch e as Exception" al conjunto de bloques Catch para que ningún error vuelva a pasar a través del control de excepciones a no ser que desee específicamente iniciar un error.
- **Iniciar errores**. Puede devolver cualquier error al llamador de forma explícita. Asimismo, puede enviar el error actual, o cualquier otro, al controlador de excepciones del llamador utilizando la instrucción **Throw**.

> **Sugerencia** Si inicia una excepción utilizando la palabra clave **Throw**, el control de errores **On Error Goto** del tipo de Visual Basic 6.0 también podrá interceptar el error. Es decir, el tiempo de ejecución de .NET utiliza la misma estructura en todas las excepciones, tanto si emplea las convenciones de control de errores antiguas como si utiliza las nuevas.

Paso de información de error

Si desea interceptar excepciones diferentes y devolverlas al llamador como un tipo de excepción simple, utilice la instrucción **Throw**, que le facilitará la tarea. En el siguiente ejemplo, el código detecta todas las excepciones e, independientemente de la causa de la excepción, devuelve un objeto **FileNotFoundException** al llamador. En determinados casos, como el nuestro, el procedimiento de llamada probablemente no esté interesado en lo que ocurrió ni en la razón por la que el archivo no se

ha podido encontrar. Al llamador sólo le interesa que el archivo no está disponible y que necesita distinguir dicha excepción en particular de otras diferentes.

Constructor del objeto Exception

El constructor del objeto **Exception** se puede sobrecargar de varios modos. No pasando ningún parámetro (en cuyo caso obtendrá un objeto **Exception** genérico, con valores predeterminados para las propiedades), pasando una cadena que indique un mensaje de error que se devolverá al llamador, o pasando una cadena y un objeto **Exception**, indicando el mensaje de error y la excepción original (rellenando la propiedad **InnerException** de la excepción que se devuelve al llamador). En este ejemplo se utiliza el constructor final y se devuelve la excepción interna.

Asimismo, puede que desee poner la información de la excepción original a disposición del llamador, junto con la excepción generada por el código. En ese caso, el constructor de la clase Exception incluye una versión sobrecargada que permite especificar la excepción interna. Es decir, puede pasar el objeto Exception que produjo originariamente el error. El llamador puede investigar esta excepción si es necesario.

> **Sugerencia** La propiedad **InnerException** de una excepción es en sí un objeto **Exception** y puede incluir también una propiedad **InnerException** que no sea **Nothing**. Por tanto, cuando comience a utilizar la propiedad **InnerException**, puede que acabe siguiendo una lista vinculada de excepciones. Asimismo, puede que deba seguir recuperando la propiedad **InnerException** repetidamente hasta que devuelva **Nothing** para examinar todos los errores que puedan haber ocurrido.

En el siguiente ejemplo, el procedimiento **TestThrow** devuelve la excepción **FileNotFoundException** al llamador correspondiente, independientemente del error que reciba. A continuación, rellena la propiedad **InnerException** de la excepción con el objeto Exception original. Este ejemplo también muestra el mensaje de error generado junto con el texto asociado a la excepción original:

```
' Opción Throw Exception del formulario de ejemplo.
Private Sub ThrowException()
  Dim lngSize As Long
  Dim s As FileStream

  ' Detectar una excepción iniciada por el procedimiento llamado.
  Try
    TestThrow()
  Catch e As FileNotFoundException
    MessageBox.Show("Error: " & e.Message)
    ' Utilizar e.InnerException para obtener el error
    ' que desencadenó este error.
    MessageBox.Show(e.InnerException.Message)
  End Try
End Sub

Private Sub TestThrow()
  Dim lngSize As Long
  Dim s As FileStream

  ' Independientemente de la causa del error, devolver
```

```
  ' la excepción de archivo no encontrado.
  Try
    s = File.Open(txtFileName.Text, FileMode.Open)
    lngSize = s.Length
    s.Close()
  Catch e As Exception
    Throw (New FileNotFoundException( _
      "No se puede abrir el archivo especificado.", e))
  End Try
End Sub
```

Ejecución incondicional de código

Puede que, además del código de los bloques **Try** y **Catch**, desee agregar código que se ejecute independientemente de si se produce o no un error. Es posible que necesite liberar recursos, cerrar archivos o controlar otro tipo de aspectos que deban tener lugar en cualquier circunstancia. Para ejecutar código de forma incondicional, utilice el bloque **Finally**.

El bloque Finally

Para ejecutar código de forma incondicional, agregue un bloque **Finally** después de cualquier bloque **Catch**. El código de este bloque se ejecuta si su código inicia una excepción e incluso si agrega una instrucción **Exit Function** (o **Exit Sub**) explícita dentro de un bloque **Catch**. Por su parte, el código del bloque **Finally** se ejecuta después del código de control de excepciones pero antes de que el control vuelva al procedimiento que realizó la llamada.

Por ejemplo, puede que crea necesario que el código defina la variable de objeto **FileStream** como **Nothing**, independientemente de si se produce o no un error durante el uso del archivo. Puede modificar el procedimiento del modo siguiente, llamando al código de finalización tanto si se produce un error como si no:

```
' Opción Test Finally del formulario de ejemplo.
Private Sub TestFinally()
  Dim lngSize As Long
  Dim s As FileStream

  Try
    s = File.Open(txtFileName.Text, FileMode.Open)
    lngSize = s.Length
    s.Close()
  Catch e As Exception
    MessageBox.Show(e.Message)
  Finally
    ' Ejecutar este código independientemente de lo que ocurra.
    s = Nothing
  End Try
End Sub
```

> **Sugerencia** Aunque el bloque **Try**/**End Try** deba incluir uno o varios bloques **Catch**, o un bloque **Finally**, no es necesario que incluya ambos. Es decir, no importa que un bloque **Finally** no incluya bloques **Catch**. ¿Por qué se debe incluir un bloque **Finally** aunque no

se incluya un bloque **Catch**? Si el procedimiento genera una excepción, el tiempo de ejecución de .NET buscará un controlador de excepciones adecuado. Esto puede implicar que el tiempo de ejecución deje al procedimiento buscando dicho controlador de excepciones en la pila de llamadas (algo que ocurrirá con toda seguridad si no hay ningún bloque **Catch**). Incluya un bloque **Finally** si desea ejecutar código antes de que el tiempo de ejecución deje el procedimiento. Si, por ejemplo, trabaja con un objeto que incluye un método **Dispose** y desea asegurarse de que lo llama antes de dejar el procedimiento, sitúe dicha llamada en el método **Dispose** en un bloque **Finally**, tanto si utiliza un bloque **Catch** como si no. De ese modo, incluso si ocurre un error, la llamada del método **Dispose** tendrá lugar antes de que .NET Framework devuelva la excepción al procedimiento que realizó la llamada.

Creación de clases Exception

.NET Framework no siempre proporciona la clase Exception que se ajusta a sus necesidades. Tal vez desee generar una excepción siempre que el usuario seleccione un archivo de tamaño superior a 100 bytes. Aunque esto no se considera una condición de excepción, puede que constituya una condición de error dentro de su aplicación.

Para crear su propia clase de excepciones, siga estos pasos:

1. Cree una clase nueva.
2. Herede de la clase base ApplicationException.

   Nota Puede heredar de cualquier clase que herede de la clase Exception. Por ejemplo, puede heredar de IOException o FileNotFoundException. Cualquiera de ellas actuará como clase base de su propia excepción. No obstante, la documentación recomienda que no se herede directamente de Exception.

3. Proporcione su propio método **New** (agregue las sobrecargas adecuadas, según sea necesario). Vuelva a llamar a MyBase.New para incluir la llamada en el constructor de la clase base.
4. Agregue la funcionalidad que necesite.

La clase FileTooLargeException

El proyecto de ejemplo incluye la siguiente definición de clase (en el módulo frmErrors.vb), así como la de la clase FileTooLargeException:

```
Public Class FileTooLargeException
  Inherits ApplicationException
  Private mlngFileSize As Long

  Public Sub New(ByVal Message As String)
    MyBase.New(Message)
  End Sub

  Public Sub New(ByVal Message As String, _
   ByVal Inner As Exception)
    MyBase.New(Message, Inner)
```

```
  End Sub

  Public Sub New(ByVal Message As String, _
   ByVal Inner As Exception, ByVal FileSize As Long)
    MyBase.New(Message, Inner)
    mlngFileSize = FileSize
  End Sub

  Public ReadOnly Property FileSize() As Long
    Get
      Return mlngFileSize
    End Get
  End Property
End Class
```

Esta clase ofrece las propiedades estándar de la clase Exception (ya que hereda de ApplicationException), pero agrega un nuevo elemento: un constructor que permite pasar el tamaño del archivo que desencadenó la excepción. Asimismo, incluye la propiedad **FileSize**, de modo que los llamadores de los procedimientos pueden determinar el tamaño del archivo que desencadenó la excepción.

La función **GetSize**, que se muestra a continuación, intenta abrir un archivo. Si el tamaño del archivo solicitado es demasiado grande, **GetSize** devuelve FileTooLargeException a su llamador correspondiente, pasando su propio mensaje de error y el tamaño del archivo solicitado:

```
Private Function GetSize( _
 ByVal strFileName As String) As Long
  Dim lngSize As Long
  Dim s As FileStream

  ' Devolver el tamaño del archivo. Si es superior a 100 bytes
  ' (tamaño arbitrario), iniciar FileTooLargeException
  ' (excepción definida por el usuario) al llamador.

  Try
    s = File.Open(txtFileName.Text, FileMode.Open)
    lngSize = s.Length
    s.Close()
    If lngSize > 100 Then
      ' Devolver la nueva excepción. No hay
      ' excepción interna que devolver; pasar Nothing.
      Throw (New FileTooLargeException( _
       "El archivo seleccionado es demasiado grande.", _
       Nothing, lngSize))
    End If
    Return lngSize
  Catch
    ' Devolver la excepción al llamador.
    Throw
  Finally
    ' Ejecutar este código independientemente de lo que ocurra.
    s = Nothing
  End Try
End Function
```

El procedimiento de prueba pasa el archivo especificado del formulario de ejemplo e intercepta FileTooLargeException. En este bloque **Catch** específico, el código recupera la propiedad **FileSize** de la excepción, se compila y se ejecuta correctamente (incluso si un objeto **Exception** normal no incluye una propiedad **FileSize**) porque esta excepción concreta, FileTooLargeException, sí que incluye esta propiedad:

```
' Opción User-Defined Exception del formulario de ejemplo.
Private Sub UserDefinedException()
  Dim lngSize As Long

  ' Probar una excepción definida por el usuario.

  Try
    lngSize = GetSize(txtFileName.Text)
  Catch e As FileTooLargeException
    MessageBox.Show( _
     String.Format( _
     "Seleccione un archivo de menor tamaño. " & _
     "El archivo seleccionado era de {0} bytes.", _
     e.FileSize))
  Catch e As Exception
    MessageBox.Show(e.Message)
  End Try
End Sub
```

> **Sugerencia** Encontrará útil poder crear sus propias clases de excepciones, heredando de la clase base ApplicationException siempre que requiera agregar su propia información a la información estándar de un error determinado. Tal vez desee crear una clase de excepción que proporcione información completa sobre el marco de pila (una estructura de datos que contiene la pila de llamadas), en lugar de la cadena simple que incluye .NET Framework en la propiedad **StackFrame**. Para ello, puede utilizar la clase StackTrace y sus miembros correspondientes. Si desea obtener más información sobre las clases StackTrace y StackFrame, consulte la documentación de .NET Framework.

Resumen

- El control de excepciones estructurado aporta una mayor eficacia que el mecanismo de control de errores de Visual Basic 6.0.
- Utilice un bloque a **Try** para agregar código de control de excepciones a un bloque de código.
- Agregue bloques **Catch**, según sea necesario, para interceptar excepciones individuales.
- El tiempo de ejecución de .NET controla los bloques **Catch** en orden, en busca de una coincidencia "es una" en la excepción actual y utiliza el primer bloque que coincida.
- Puede anidar bloques **Try**, lo que facilita la tarea de insertar y extraer estados de control de excepciones de forma efectiva.
- Agregue un bloque **Finally** al bloque **Try** para ejecutar el código de forma incondicional, independientemente de si se produce o no un error.
- Puede crear sus propias clases de excepciones que hereden de la clase base Exception (o cualquier clase que herede de la misma) para agregar su propia funcionalidad.

Acerca del autor

Ken Getz es un consultor de gran experiencia de MCW Technologies que dedica su tiempo a la programación, la escritura y la formación. Es especialista en herramientas y aplicaciones escritas en Microsoft Access, Visual Basic y en el resto de programas de Office y BackOffice. Es coautor de varios libros, entre los que se incluyen *Access 97 Developer's Handbook* (en inglés) con Paul Litwin y Mike Gilbert, *Access 2000 Developer's Handbooks* (en inglés) con Paul Litwin y Mike Gilbert, *Access 2002 Developer's Handbooks* (en inglés) con Paul Litwin y Mike Gunderloy, *Visual Basic Language Developer's Handbook* (en inglés) con Mike Gilbert y *VBA Developer's Handbook* (en inglés) con Mike Gilbert (Sybex). Asimismo, es coautor de diverso material de formación e imparte clases para la empresa AppDev. Es ponente de conferencias técnicas y desde 1994 ha participado en las conferencias de Microsoft Tech*Ed. Colabora como editor técnico en la revista *Access/VB/SQL Advisor* (en inglés) y en la edición de la revista *Microsoft Office Solutions* (en inglés) de Informant Communication Group.

Acerca de Informant Communications Group

Informant Communications Group, Inc. (www.informant.com) es una empresa multimedia diversificada centrada en el sector de las tecnologías de la información. ICG se fundó en 1990 y está especializada en las publicaciones, conferencias, publicación de catálogos y sitios Web de desarrollo de software. Con oficinas en los Estados Unidos y el Reino Unido, ICG es una empresa de prestigio conocida por sus sistemas de integración de contenido de marketing y multimedia, que satisface la demanda creciente de profesionales de tecnologías de la información capaces de ofrecer calidad en la información tecnológica.

Edición técnica: PDSA, Inc. o KNG Consulting